KaSim reference manual v1.05

J. Krivine

November 8, 2010

This document is work in progress...

Contents

Chapter 1

Introduction

1.1 Preamble

This manual contains a description of the usage of KaSim. Although it contains a brief description of Kappa, it is *not* intended as a Kappa tutorial. Therefore, in the following of this manual some familiarity with Kappa is assumed and we let the reader refer to `http://KappaLanguage.org` for further explanations.

1.2 The KaSim engine

KaSim is an open source stochastic simulator of rule-based models [3, 2, 4] written in the κ-calculus. Basically KaSim takes one or several kappa files as input and generates stochastic trajectories of various observables. KaSim implements Danos and Krivine's network free simulation algorithm [1] that adapts Gillespie's algorithm [5, 6] for rule-based models.

A *simulation event* corresponds to the application of a rewriting rule, contained in the kappa file, to the current graph (also called a *mixture*). The rule is selected according to its *activity*, *i.e* the number of instances it has in the current mixture, multiplied by its kinetic rate, and applied one of its possible instance in the graph. It results in a new graph together with an updated activity for all rules (see Fig. 1.1).

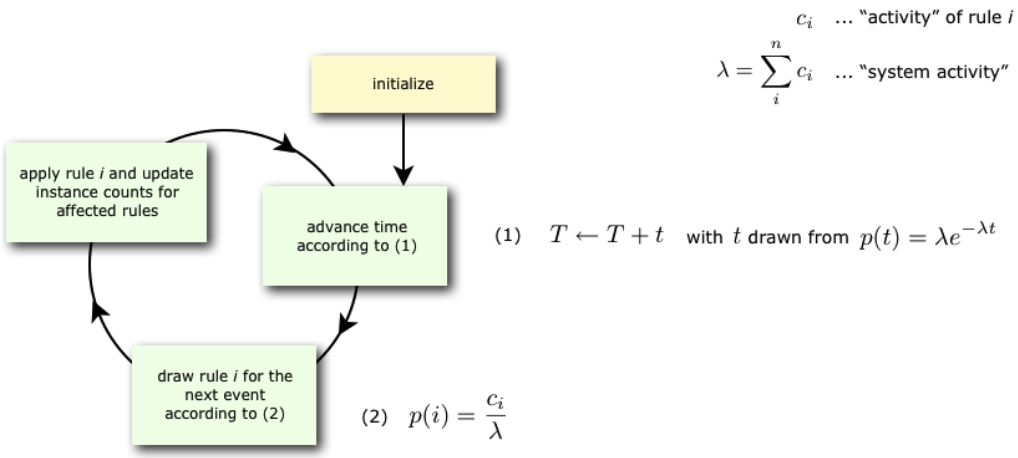

Figure 1.1: The event loop

Importantly, the cost of such an event is independent of the size of the graph it is applied to [1]. Note that KaSim is not equipped with a curve visualization tool. However, the outputted data are in text format and usable with any standard plotting software such as GnuPlot.

1.3 Support

- Kappa language tutorials and downloads: `http://kappalanguage.org`
- Bug reports should be posted on github: `https://github.com/jkrivine/KaSim/issues`
- Questions and answers on the kappa-user mailing list: `http://groups.google.com/group/kappa-users`
- Want to contribute to the project? `jkrivine` at `pps` dot `jussieu` dot `fr`

Chapter 2

Installation

2.1 Obtaining the sources

To obtain KaSim you can either use pre-compiled binaries available on KappaLanguage.org or compile the sources for your architecture. To do so, download the source code from `https://github.com/jkrivine/KaSim` and make sure you have a recent ocaml compiler installed. From a terminal window type `ocamlopt.opt -v`. If nothing appears then you need to install Ocaml Native compiler that can be downloaded from `http://caml.inria.fr/download.en.html`.

2.2 Compilation

Once Ocaml is safely installed, untar KaSim archive and compile following these few steps:

```
$ tar xzvf kasim.tar.gz -d Kappa
$ cd Kappa
$ make
```

At the end of these steps you should see, in the `Kappa` directory, an executable file name KaSim. In order to check the compilation went fine, simply type `.\KaSim -version`. If the ocaml native compiler `ocamlopt.opt` is not in the path of your system, you may set the variable `OCAMLBINPATH` to point to the location of the compiler by editing the corresponding line in the Makefile.

Chapter 3

The command line

3.1 General usage

From a terminal window, KaSim can be invoked by typing

```
$ KaSim -i file_1 ...  -i file_n [option]
```

where `file_i` are the input kappa files containing the rules, initial conditions and observable see Chapter 4 below. Tables 3.1 and 3.2 summarize all the options that can be given to the simulator. Basically one should specify an upper bound either in bio time (arbitrary time unit), or number of events. Note that bio-time is computed using Gillespie's formula for time advance (see Fig. 1.1) and should not be confused with CPU-time (it's not even proportional). In doubt we recommend using a bound in number of events since the cost of one event application is bounded (in CPU time) by a constant, so the simulation time of n events is roughly k times faster than a simulation of $k * n$ events.

3.2 Main options

Table 3.1: Command line: main options

Argument	Description
`-e` e_{max}	Terminates simulation after $e_{max} \geq 0$ events
`-t` t_{max}	Terminates simulation after $t_{max} \geq 0.0$ time units
`-p` n	Produces a data file (default: `data.out`) with $n \geq 0$ data points
`-o` *file*	Set the name of data file to *file*
`-d` *dir*	Output any produced file to the directory *dir*

3.3 Advanced options

Table 3.2: Command line: advanced options

Argument	Description
`-seed` n	Seeds the pseudo-random number generator $n > 0$
`-implicit-signature`	Automatically deduce agent signatures (see Chapter 6)

Example

The command

```
$ KaSim -i model.ka -e 1000000 -p 1000 -o model.out
```

will generate a file `model.out` containing the trajectories of the observables defined in the kappa file `model.ka`. The file `model.out` will contain 1000 data points (*i.e* in this case, a measure will be taken every 1000 events). The command

```
$ KaSim -i init.ka -i rules.ka -i obs.ka -i mod.ka -t 1.5 -p 1000
```

will generate a file `data.out` (default name) containing 1000 data points of a simulation of 1.5 seconds (arbitrary time units) of the model. Note that the input kappa file is splited in 4 files containing, for instance, the initial conditions, `init.ka`, the rule set, `rules.ka`, the observables, `obs.ka`, and the perturbations, `pert.ka` (refer to Chapter 4 for details). The order in which the files are given does not matter.

Chapter 4

The kappa file

4.1 General remarks

The *Kappa File* (KF) is the formal representation of your model. We use KF to denote the union of the files that are given as input to KaSim (argument `-i`). Each line of the KF is interpreted by KaSim as a *declaration*. If the line is ended by the escape character '\' the continuation of the declaration is parsed onto the next line. Declarations can be of 4 types: *signatures* (Sec. 4.2), *rules* (Sec. 4.3), *variables* (Sec. 4.4), *initial conditions* (Sec. 4.5) and *perturbations* (Sec. 6.1). The KF's structure is quite flexible and it can be divided in any sub-files in which the order of declarations does not matter. Comments can be used by inserting the marker `#` that tells KaSim to ignore the rest of the line.

4.2 Agent signature

Agent signatures constitute a form of typing information about the agents that are used in the model. It contains information about the name and number of interaction sites the agent has, and about their possible internal states. A signature is declared in the KF by the following line:

`%agent:` *signature_expression*

according to the grammar given Table 4.1. For instance the line:

```
%agent: A(x,y~u~p,z~0~1~2) # Signature of agent A
```

will declare an agent `A` with 3 *(interaction) sites* `x`,`y` and `z` with the site `y` possessing two *internal states* `u` and `p` (for instance for the unphosphorylated and phosphorylated forms of `y`) and the site `z` having possibly 3 states respectively 0, 1 and 2. Note that internal states values are treated as untyped symbols by KaSim, so choosing a character or an integer as internal state is purely matter of convention.

signature_expression	::=	*Id* (*sig*)
sig	::=	*Id internal_state_list*, *sig* | ε
internal_state_list	::=	~*Id internal_state_list* | ε

Table 4.1: Agent signature expression: terminal symbol are denoted in blue. Symbol *Id* can be any string generated by regular expression [*a-z A-Z* 0-9][*a-z A-Z* 0-9 _ −]*. Terminal symbol ε stands for the empty symbol.

4.3 Rules

Once agents are declared, one may add to the KF the rules that describe their dynamics through time. Roughly a Kappa rule looks like

`'my rule'` *kappa_expression* → *kappa_expression* `@` γ

where `'my rule'` can be any name that will refer to the subsequent rule that can be decomposed into a *left hand side* (LHS) and a *right hand side* (RHS) kappa expressions together with a *kinetic rate* k. Kappa expressions are generated by the grammar given Table 4.2.

kappa_expression	::=	*agent_expression* , *kappa_expression* | ε
agent_expression	::=	*Id* (*interface*)
interface	::=	ε | *site internal_state link_state*
internal_state	::=	ε |~*Id*
link_state	::=	ε | !*n* | !_ | ?

Table 4.2: Kappa expressions: In addition to the conventions of Table 4.1, symbol n denotes any positive integer.

4.3.1 A simple rule

With the signature of `A` defined in the previous section, the line

`'A dimerization' A(x),A(y~p)` → `A(x!1),A(y~p!1) @` γ

denotes a dimerization rule between two instances of agent `A` provided the second is phosphorylated (say that is here the meaning of `p`) on site `y`. Note that the bond between both `A`s is denoted by the identifier `!1` which uses an arbitrary integer (`!0` would denote the same bond. In Kappa, a bond may connect exactly 2 sites so any occurrence of a bond identifier `!`n has to be paired with exactly one other sibling in the expression. Note also the fact that site `z` of `A` is not mentioned in the expression which means that it has no influence on the triggering of this rule. This is the *don't care don't write convention* (DCDW) that plays a key role in resisting combinatorial explosion when writing models.

4.3.2 Adding and deleting agents

Sticking with A's signature, the rule

'budding A' A(z) → A(z!1),A(x!1) @ γ

indicates that an agent A free on site z, no matter what its internal state is, may beget a new copy of A bound to it *via* sites x. Note that in the RHS, agent A's interface is not completely described. Following the DCDW convention, KaSim will then assume that the sites that are not mentioned are created in the *default state*, *i.e* the appear free of any bond and their internal state (if any) is the first of the list shown in the signature (here state u for y and 0 for z).

Importantly, KaSim respects the *longest prefix convention* to determine which agent in the RHS stems from an agent in the LHS. In a word, from a rule of the form $a_1, \ldots, a_n \to b_1, \ldots, b_k$, with a_is and b_js being agents, one computes the biggest indices $i \leq n$ such that the agents $a_1, \ldots, a_i$ are pairwise consistent with $b_1, \ldots, b_i$, *i.e* the a_js and b_js have the same name and the same number of sites. In which case we say that the for all $j \leq i$, a_j is *preserved* by the transition and for all $j > i$, a_j is *deleted* by the transition and b_j is *created* by the transition. This convention allows us to write a deletion rule as:

'deleting A' A(x!1),A(z!1) → A(x) @ γ

4.3.3 Side effects

It may happen that the application of a rule has some *side effects* on agents that are not mentioned explicitly in the rule. Consider for instance the previous rule:

'deleting A' A(x!1),A(z!1) → A(x) @ γ

The A in the graph that is matched to the second occurrence of A in the LHS will be deleted by the rule. As a consequence all its sites will disappear together with the bonds that were pointing to them. For instance, when applied to the graph

G =A(x!1,y~p,z~2),A(x!2,y~u,z~0!1),C(t!2)

it results in a new graph G' = A(x!1,y~p,z~2),C(t) where the site t of C is freed as side effect.

Wildcard symbols for link state ? (for bound or not), !_ (for bound to someone), may also induce side effects when they are not preserved in the RHS of a rule, as in

'Disconnect A' A(x!_) → A(x) @ γ

or

'Force bind A' A(x?) → A(x!1),C(t!1) @ γ

Both these rule will cause KaSim to raise a warning at rule compile time.

4.3.4 Kinetic rates

Rules are equipped with a *kinetic rate*, denoted by γ above. For KaSim, this number should correspond to the *stochastic rate constant* at which the corresponding reaction should be applied. We let the reader refer to `http://www.KappaLanguage.org` for a more complete tutorial on kinetic rates, but basically γ corresponds to the *deterministic rate constant* k of the reaction corrected by the volume V in which the model is considered.

We often want to express γ in "per molecule" rather than "per mol". This only changes the numerical value, not the dimension (both are numbers of molecules):

$$\gamma = \frac{k}{(\mathcal{A}\, V)^{(a-1)}} \qquad [molecule^{-1}s^{-1}], \tag{4.1}$$

where $\mathcal{A} = 6.022 \cdot 10^{23}$ is Avogadro's number, k in molar units $M^{-1}s^{-1}$ and $a > 0$ is the arity of the rule (*i.e* 2 for a bimolecular rule)

- Mammalian cell: $V = 2.25\ 10^{-12}l$ ($1l = 10^{-3}m^3$), and $\mathcal{A}V = 1.35\ 10^{12}$.

  A concentration of $1M$ in a mammalian cell volume corresponds to $1.35\ 10^{12}$ molecules; $1nM \approx 1350$ molecules per cell.

- Yeast cell (haploid): $V = 4\ 10^{-14}l$, and $\mathcal{A}V = 2.4\ 10^{10}$.

  A concentration of $1M$ in a yeast cell volume corresponds to $2.4\ 10^{10}$ molecules; $1nM \approx 24$ molecules per cell. The volume is doubled in a diploid cell.

- E.coli cell: $V = 10^{-15}l$, and $\mathcal{A}V = 10^8$.

  A concentration of $1M$ in a yeast cell volume corresponds to 10^8 molecules; $10nM \approx 1$ molecule per cell.

The following table lists a few ballpark figures for deterministic rate constants and their stochastic counterparts in a mammalian cell volume:

process	k	γ	stoch. dimension
general binding	$10^7 - 10^9$	$10^{-5} - 10^{-3}$	$molecule^{-1}s^{-1}$
general unbinding	$10^{-3} - 10^{-1}$	$10^{-3} - 10^{-1}$	s^{-1}
dephosphorylation	1	1	s^{-1}
phosphorylation	0.1	0.1	s^{-1}
receptor dimerization	$2\ 10^6$	$1.6\ 10^{-6}$	$molecule^{-1}s^{-1}$
receptor dissociation	$1.6\ 10^{-1}$	$1.6\ 10^{-1}$	s^{-1}

4.4 Variables

In the KF it is also possible to declare *variables* whose value may range over various *observables* of the model. A variable is declared by a line of the form

`%var: 'var_name' :=` *variable_expression* | *kappa_expression*

where `var_name` can be any string and *variable_expression* is any algebraic expression on variable names (other than `var_name`) using predefined operators summarized in Table 4.3.

Symbol	Interpretation
`[E]`	the number of simulation events since the beginning of the simulation
`[T]`	the bio-time of the simulation
`'v'`	the value of variable 'v'
`[f]`	the intuitive mathematical function or constant associated to $f \in \{$`log,sin,cos,tan,sqrt,pi`$\}$
`[inf]`	symbol for ∞
`[modulo]`	the *modulo* operator
`[exp]`	the exponentiation operation $x \mapsto e^x$
`[abs]`	the integer part $x \in \mathbb{R} \mapsto \lfloor x \rfloor \in \mathbb{N}^+$
`+,-,*,/,^`	the corresponding mathematical operators

Table 4.3: Symbol usable in variable expressions.

For instance the declarations

```
%var: 'homodimer' := A(x!1),A(x!1)

%var: 'aa' := 'homodimer'/2
```

define 2 variables, the first one tracking the number of embeddings of `A(x!1),A(x!1)` in the graph over time, while the second divides this value by 2 (the number of automorphisms in `A(x!1),A(x!1)`. Note that it is possible to use algebraic expressions as kinetic rates as in

```
%var: 'k_on' := 10.E-6 # per molecule per second

'ab' A(x),A(x) -> A(x!1),A(x!1) @ 'k_on'/2
```

KaSim may output values of variables in the data file (see option `-p` in Chapter 3) using plot declaration:

```
%plot: 'var_name'
```

One may use the shortcut:

`%obs: 'var_name' :=` *variable_expression*

to declare a variable and at the same time require it to be outputted in the data file.

4.5 Initial conditions

The initial mixture to which rules in the KF will be applied are declared as

`%init:` *n kappa_expression*

where $n > 0$ can be any integer. This will add to the initial state of the model n copies of the graph described by the kappa expression. Again the DCDW convention allows us not to write the complete interface of added agents (the remaining sites will be completed according to the agent's signature). For instance:

```
%init:  1000 (A,A(y p))
```

will add 1000 instances of `A` in its default state `A(x,y~u,z~0)` and 1000 instances of `A` in state `A(x,y~p,z~0)`. Note that `A` is equivalent to writing `A()`. As any other declaration, `%init` can be used multiple times, and agents will add up to the initial state.

Chapter 5

A simple model

We describe below the content of a simple Kappa model and give examples of some typical run.

5.1 ABC.ka

```
1. #### Signatures
2. %agent:  A(x,c) # Declaration of agent A
3. %agent:  B(x) # Declaration of B
4. %agent:  C(x1~u~p,x2~u~p) # Declaration of C with 2 modifiable sites
5. #### Rules
6. 'a.b' A(x),B(x) -> A(x!1),B(x!1) @ 'on_rate' #A binds B
7. 'a..b' A(x!1),B(x!1) -> A(x),B(x) @ 'off_rate' #AB dissociation
8. 'ab.c' A(x!_,c),C(x1~u) ->A(x!_,c!2),C(x1~u!2) @ 'on_rate' #AB binds C
9. 'mod x1' C(x1~u!1),A(c!1) ->C(x1~p),A(c) @ 'mod_rate' #AB modifies x1
10. 'a.c' A(x,c),C(x1~p,x2~u) -> A(x,c!1),C(x1~p,x2~u!1) @ 'on_rate' #A binds C on x2
11. 'mod x2' A(x,c!1),C(x1~p,x2~u!1) -> A(x,c),C(x1~p,x2~p) @ 'mod_rate' #A modifies x2
12. #### Variables
13. %var:  'on_rate':= 1.0E-4 # per molecule per second
14. %var:  'off_rate':= 0.1 # per second
15. %var:  'mod_rate':= 1 # per second
16. %obs:  'AB' A(x!x.B)
17. %obs:  'Cuu' C(x1~u,x2~u)
18. %obs:  'Cpu' C(x1~p,x2~u)
19. %obs:  'Cpp' C(x1~p,x2~p)
20. #### Initial conditions
```

```
21. %init:  1000 A,B
22. %init:  10000 C
```

Line 1-4 of this KF contains signature declaration. Agents of type `C` have 2 sites `x1` and `x2` whose internal state may be `u`(nphosphorylated) or `p`(hosphorylated). Recall that the default state of these sites is `u` (the first one). Line 8, rule `'ab.c'` binds an `A` connected to someone on site `x` (link type `!_`) to a `C`. Note that the only rule that binds an agent to `x` of `A` is `'a.b'` at line 6. Hence the use of `!_` is a commodity and the rule could be replaced by

`'alt_ab.c' A(x!1,c),B(x!1),C(x1~u)` $\rightarrow \ldots$

There are two main points to notice about this model: `A` can modify both sites of `C` once it is bound to them. However, only an `A` bound to a `B` can connect on `x1` and only a free `A` can connect on `x2`. Note also that `x2` is available for connection only when `x1` is already modified.

5.2 Some runs

We try first a coarse simulation of $100,000$ events (10 times the number of agents in the initial system).

```
$ KaSim -i ABC.ka -e 100000 -p 1000 -o abc.out
```

Plotting the content of the `abc.out` file one notices that nothing of significant interest happen to the observables after 250s. So we can now specify a meaningful time limit by running

```
$ KaSim -i ABC.ka -e 100000 -t 250 -o abc.out
```

which produces the data points whose rendering is given Fig. 5.1. We will use this model as a running example for the next chapter, in order to illustrate various advanced concepts.

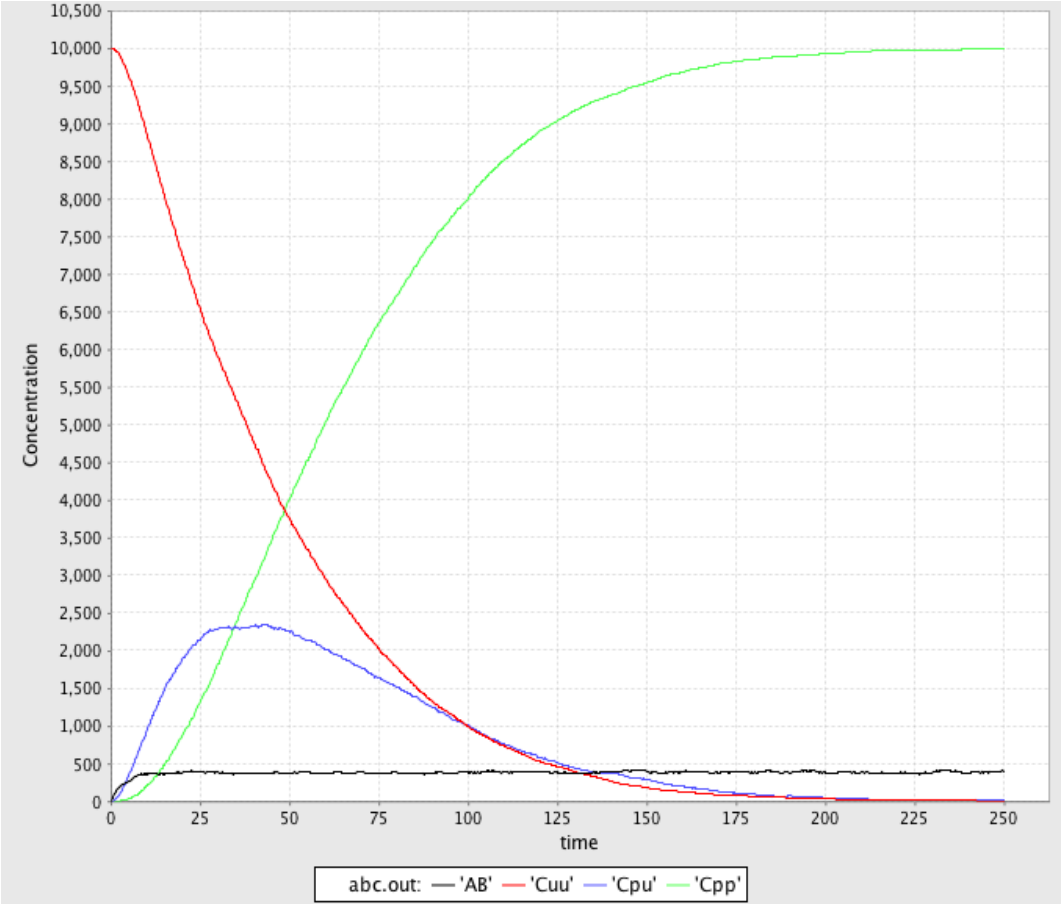

Figure 5.1: Simulation of the ABC model: population of unmodified `Cs` (observable `Cuu` in red) drops rapidly and is replaced, in a first step by simply modified `Cs` (observable `Cpu` in blue) which are in turn replaced by doubly modified `Cs` (observable `Cpp` in red). Note that the population of `AB` complexes (observable `AB` in black) stabilizes slightly below 400 individuals after about 20s.

Chapter 6

Advanced concepts

6.1 Perturbation language

6.2 Link type

6.3 Implicit signature

Bibliography

[1] Vincent Danos, Jérome Féret, Walter Fontana, and Jean Krivine. Scalable simulation of cellular signaling networks. In *Proceedings of APLAS'07: 5^th ASIAN symposium on programming languages and systems*, volume 4807 of *LNCS*, pages 139–157, 2007. Invited paper.

[2] Vincent Danos, Jérôme Feret, Walter Fontanta, Russ Harmer, and Jean Krivine. Rule based modeling of biological signaling. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Proceedings of CONCUR 2007*, volume 4703 of *LNCS*, pages 17–41. Springer, 2007.

[3] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325, 2004.

[4] James R. Faeder, Mickael L. Blinov, and William S. Hlavacek. Rule based modeling of biochemical networks. *Complexity*, pages 22–41, 2005.

[5] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, 1976.

[6] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.